28 STONE

Our Solace-native trading platform architecture

Leveraging out of the box features of Solace PubSub+ platform to build scalable, real-time trading systems

Contents

1 Introduction

We at 28Stone have built multiple electronic trading systems and marketplaces for our clients over the span of our company history. During this time, we have seen what works and what doesn't, we've fine-tuned patterns and practices to maximize impact and productivity and have gained valuable insights on architecture of such systems.

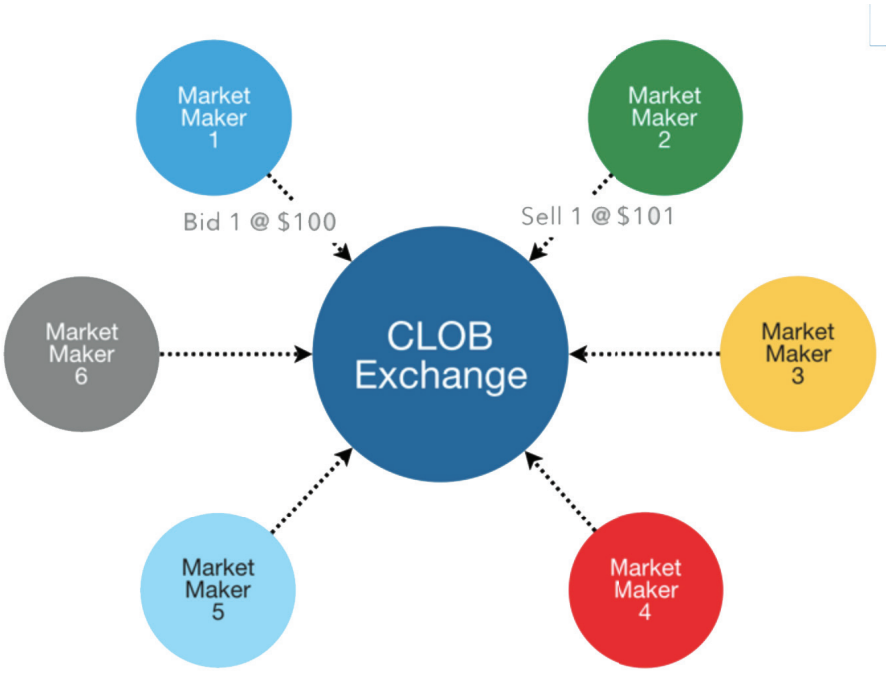
Over the years we've incrementally modernized our designs, honed our techniques and incorporated newer technologies and tools. In this paper, we describe our sample reference trading platform architecture and elaborate on how the use of the Solace PubSub+ platform has tremendously aided us.

Disclaimer

Not all trading platforms are created equal. Depending on the financial instrument and trade lifecycle, some components and flows become more important than others. Additionally, certain aspects of the design are heavily based on the system requirements and NFRs. Variations of the reference architecture blueprint described here has been successfully adapted to FX, Crypto, IR Swaps and Fixed Income trading platforms with a combination of RFQ, Auction and CLOB modalities. The reference architecture described here best suits CLOB based marketplaces.

What is a CLOB?

CLOB stands for Central Limit Order Book. In this modality, the exchange/marketplace maintains a book (also called a market) of bids and offers for instruments. The book and market depth may be fully, partially or not disclosed to participants of the exchange. The exchange matches orders of opposing sides based on price-time priority basis, either continuously or during pre-determined time-boxed trading sessions. This modality is well suited for liquid markets like equities and treasuries where there is ample order matching opportunity and wide participant engagement.



2

Reference Architecture Guidelines

Our architecture and design initiation starts with an identification of goals we strive to attain during implementation. The list below highlights important design decision criteria used during the architecture and evolution of our platform. This list is a sample only, it is by no means comprehensive.

Availability

- Services must be highly available
- Service invocation should always return some response
- Stateful services should run in a Master + N Slave (1+N) configuration
- Stateless services should be redundant

Recovery

- Stateful services must be able to recover efficiently from failure and restarts
- Services that rely on state or ordering should be able to detect missed messages

Idempotency

- Services should be able to detect and discard duplicate messages

Scalability

- Stateless services should allow for fan-out scaling
- Stateless services must allow load-balancing between instances

Deployment

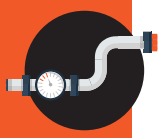
- The system should support global, multi-datacenter deployment
- Components of the system should be capable of independent deployment – all components need not be deployed for a change in a single component

Event-driven

- The system should be built on asynchronous, event-driven principles end-to-end including the front-end components

Security

- User Access Control should be native to the architecture: Avoid polluting code with entitlement conditionals
- Must use standard transport and network level security



Portability

- Message flow can be taken from one environment and replayed into another

Message model

- The message model is not monolithic and insulates from change
- Services should have the capability to be upgraded independently

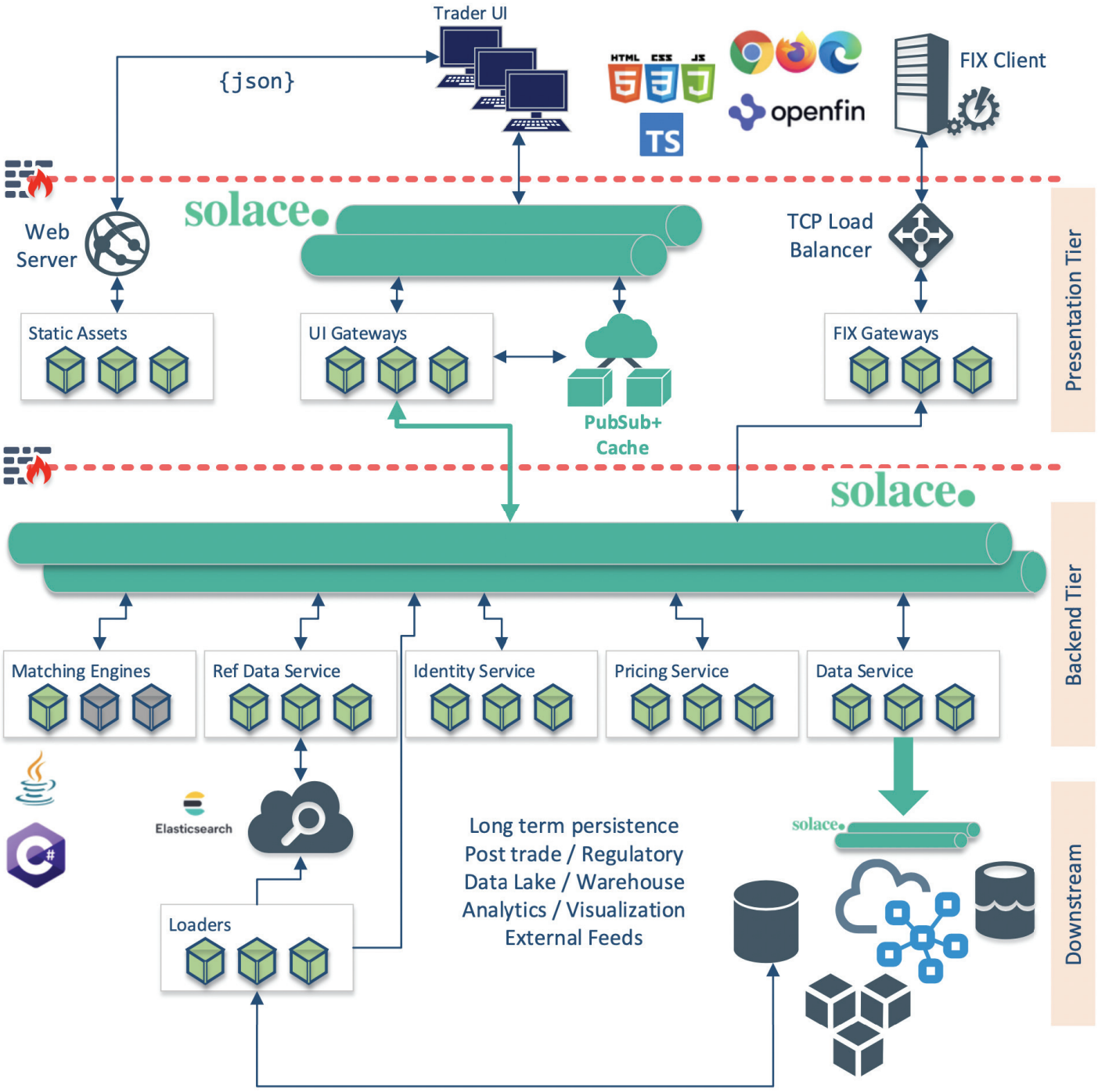
Simplicity

- Minimize and rationalize the technologies that need to be integrated

If you read between the lines, you'll see that the guidelines implicitly refer to some characteristics not listed - failover, fault tolerance, DR capability, real-time operation, replay ability and extensibility among a few.

3 Show me the Architecture!

Enough talking about it, let's now dive into the actual reference architecture blueprint.



Highlights

Before diving into the depths of the architecture, I'd like to summarize and highlight the key advantages this architecture brings to the table

Ordering is guaranteed – Solace guarantees preservation of message order which is critical for 1+N configured services such as the matching engine

Latency optimized – Solace is blazingly fast but we can further reduce end-to-end latency by utilizing Solace hardware appliances

Cloud ready – Tiers of the architecture can be hosted in hybrid environments using an Event Mesh to seamlessly extend to public cloud environments

Caching Layer – PubSub+ Cache provides a uniform, global view of the state of the market with inbuilt mechanisms to avoid race conditions between snap and livestream data requests

Extensible and Flexible – Services can be scaled, added and removed easily within a tier without impacting other tiers. The message model allows phased, backward compatible evolution

Multi-region optimized – Can be deployed in multiple geographies and linked together

QoS – Allows for both non-persistent (market data) and persistent (order flow) QoSs within a single solution

The backbone of the architecture: Solace PubSub+

In the very first iteration of the architecture several years ago, we knew we needed a messaging system. We played with a few of them, including popular open source systems. A few iterations and bake-offs later we embraced Solace PubSub+. We've seen tremendous benefit across all axes of our architectural guidelines by using Solace and it has truly become the backbone of our architecture.

You're probably thinking that modern architecture trends have shifted the arc towards service meshing with HTTP but our experiments with these architectures have shown them to be limited in terms of unified support for a variety of messaging patterns and require an undue amount of custom code and/or integrations to enable features otherwise found native to most messaging systems – for example, broadcasting, guaranteed messaging, service discovery and multiple protocol/binding support. Importantly, messaging systems are natively event-driven and steam based allowing us to take advantage of the primitives found in such platforms.

> **On a final note, in our opinion the performance, deployment flexibility and breadth of features provided out-of-the-box by Solace is unparalleled in the industry, it is simply the best messaging platform available on the market today.**

Architecture details

At the highest level the architecture is divided into two main tiers – the presentation tier and the backend tier. The functionality of the trading system is contained within these two tiers. An additional downstream tier is depicted for sake of completeness – this tier is for ancillary processes outside the core functionality of the trading system. Note that the downstream tiers use the same architecture guidelines as the trading system tiers.

One of the first things you may notice is the use of multiple Solace HA deployments. Each tier is serviced by a dedicated Solace broker HA pair. This isolates the messaging of the tier allowing tuning and optimization specific to the tier.

Presentation Tier

The presentation tier consists of a collection of 3 service groups

1.

A web farm behind an HTTPS load balancer with SSL termination, serving static assets – HTML, JavaScript and CSS

2.

A series of redundant, UI gateways that effectively implement a bridge between the UI and backend tier

3.

A FIX farm behind a TCP load balancer providing FIX based API connectivity

The presentation tier connects to the backend tier solely through Solace via the UI gateways. There are no other routes or access paths between the two tiers.

Backend Tier

The backend tier consists of a mixture of service groups in various deployment configurations depending on the supported asset class, scalability, service access pattern, and statefulness of the service provided. For example, the matching engines are 1 Master, N Slave (1+N) configured with queue based guaranteed messaging. Scheduler services are broadcast over topics. RPC style services may use topic to queue binding with non-guaranteed messaging.

4 Meeting the architecture guidelines

How does this architecture enable us to adhere to the guidelines? Let's take a deeper look

High Availability

The Solace backbone provides an extremely robust mechanism to ensure high availability of services configured in either 1+N or redundant configurations.

Stateless services use either topic or topic to queue binding based messaging. We spin up multiple, redundant service instances to achieve HA of such service groups.

For 1+N configurations we use Solace's exclusive queue binding and active flow indicator mechanism. In this model, a specified queue is configured to be exclusive which means that multiple consumers can bind to the queue but only one instance will be "active" and enabled for inbound and outbound messages. Other instances that are inactive process inbound messages, but outbound messages are suppressed. Solace internally manages the flow indicator flag and detects when a flow and therefore a master process instance has become inactive. Under these conditions another instance of the group receives the flow active indicator causing it to be the master. In our experience this has been transparent, reliable and coupled with other techniques described further in the paper results in no message loss and quick turnaround. This is the foundation of our leader election and failover process.

By simply using features provided by Solace, we achieve high availability with no custom code, races or timeout waits.

Recovery

Stateless service groups typically have no recovery requirements outside of the need to re-initialize any local caches. All data of this data is available on static PubSub+ queues.

For stateful services, we use a combination of Solace PubSub+ queues to achieve recovery in case of an instance failure. Services use a set of 3 queues such that when used in conjunction provide the necessary mechanism to restart processing from.

1.

Service (SVC) queue. This is where all the services subscriptions are to be found and the service consumes messages from this queue destructively.

2.

Recovery (RCY) queue. This has the same subscriptions as the service queue and is just consulted at start up to recover state.

3.

Last Value Queue (LVQ). As each message is processed on the service queue this queue is updated with the message ID processed – it is in effect the pointer into RCY queue to where the service is up to.

I'll take an example of 1+N configured service groups in which there is one master and N slaves. The pattern of working with these queues is

1.

When a process starts, it first attempts to read from the LVQ

a. The LVQ is exclusive, indicating only one process from a group can have an active flow.

b. This directly results in leader election – the first process to receive the flow indicator of ACTIVE is the master.

2.

It then reads the LVQ to find out where the service had previously processed to

a. If the LVQ is empty it simply starts reading from the SVC queue.

b. The normal LVQ message is COS priority 2. To see if the queue is empty, we send a COS1 sentinel message at the lower priority which is rejected when the LVQ is not empty and when read indicates it was empty – avoiding the need for a timeout.

3.

The RCY queue is then read and processed with the service being told it is in the Recovering state.

a. Any outbound messaging to the event bus is suppressed during this time.

b. The application logic processes the messages it would during normal operation.

4.

Once the message on the LVQ has been reached the process switches to destructively reading from the SVC queue.

5.

The process is then either Active if it is the master or Up if it is a secondary.

We've successfully utilized this pattern and variations of it for any service to come online and resume processing from where things last left off. Solace PubSub+ provided exclusive queue access and COS message priorities are integral to this pattern and have proven to be indispensable.

Idempotency

All services are idempotent – input messages presented to services are processed uniformly, duplicates are detected and discarded.

The essence of this works as follows - each platform entity is stamped with an ID. Every mutation of the entity generates a new sequentially increasing ID and retains the original ID as a separate field on the entity. Services performing entity mutations maintain a table of the entity's original ID to it's last amended ID. When such services are presented with a message, they lookup the last amendment ID and compare that with the ID of the entity. If the service sees that the last amendment ID is higher than the entity ID, it determines that the message is a duplicate, logs the message and suppresses further processing on the message. We also maintain an additional set of IDs that are used in idempotency around recovery scenarios but the overall approach of ID comparison remains the same.

Scalability

Scalability is inherent in the design due to the use of Solace PubSub+ and Pub/Sub messaging. Simply adding more service instances allows scaling and additional throughput. Avid readers may notice that 1+N service configurations seem to be limited by a single master process. We achieve scaling of these configurations using message sharding – dividing the input messages based on a property of the message and sending them to targeted service groups. If additional throughput is required, the Solace brokers can be scaled up to appliances or brokers could be configured to form an event mesh across multiple hosting locations using Dynamic Message Routing (DMR).

The presentation tier caches are globally meshed so regardless of where the matching occurs, all caches, across geographies have the full state of the market.

Deployment

Our architecture purposefully separates the presentation tier from the backend tier. In the presentation tier, we use PubSub+ Cache which seamlessly caches messages from subscribed topics and provides an API to read messages and subscribe to subsequent cache updates. PubSub+ Cache maintains the entire live state of the market whilst the Solace bus pushes messages directly to UIs.

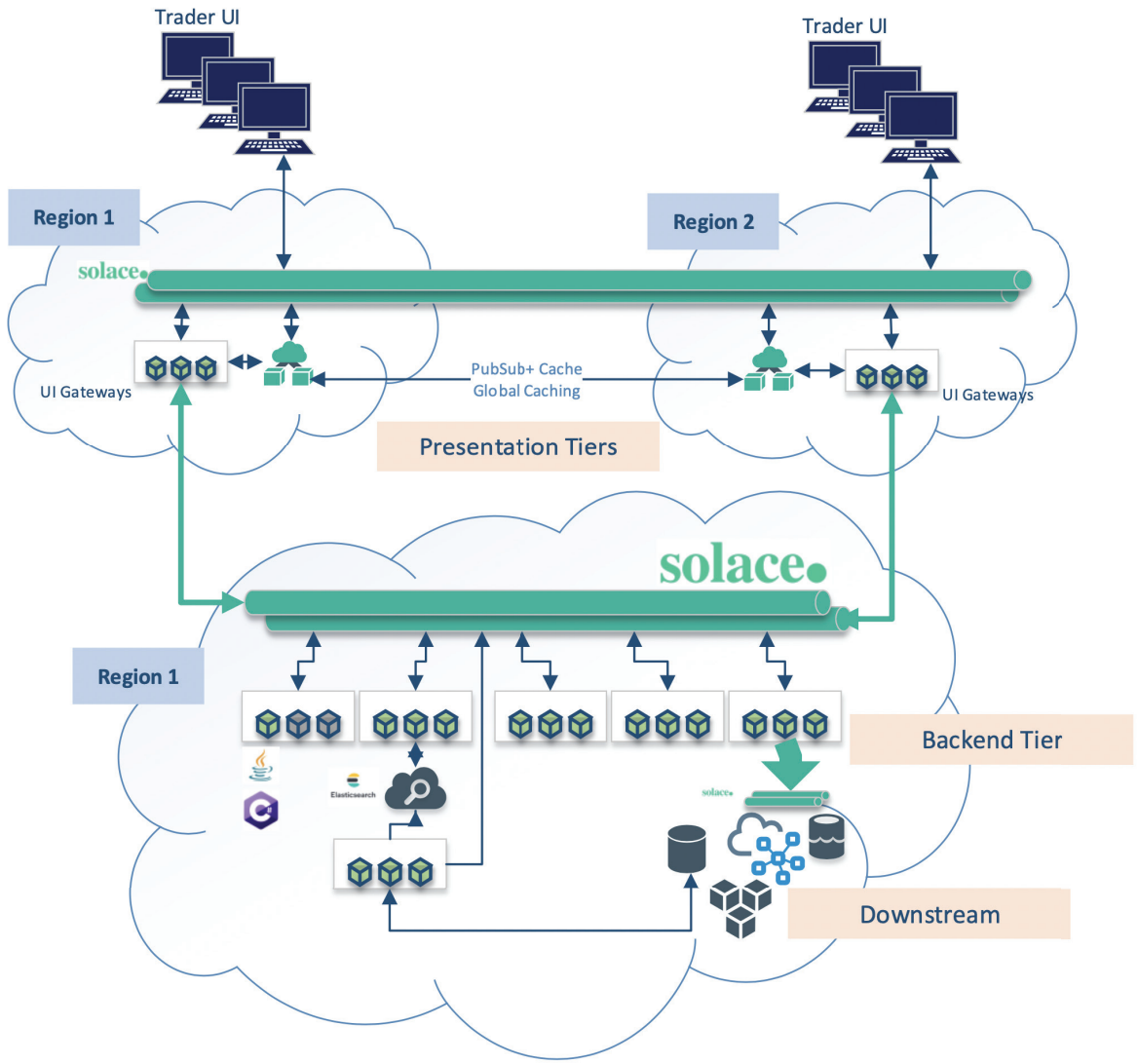
> **As global usage grows, the presentation tier may be deployed in disparate, geographically separated data centers or cloud providers.**

Geo-DNS would route connecting users to the closest presentation tier. Using the global caching features of PubSub+ Cache allows us to link distributed cache clusters and keep them in sync simply by updating the cache configuration – there is no additional routing, synchronization or clustering code required.

In most of our deployments there is a single backend tier resulting in long haul messaging across geographically separated presentation tiers. While this does increase latency for backend command processing, we have not experienced crippling service degradation because of this. Finally, since the presentation tier continually updates the UI with the state of the market UIs remain dynamic and snappy.

In terms of messaging formats – we use JSON in the presentation tier and binary Google protobuf on the backend tier. Both these formats are forward and backward compatibility friendly – allowing us to upgrade service groups independent of others.

The architecture is agnostic to location of deployment – whether it's cloud or on-prem or hybrid. Some configurations and components may change but we've used this architecture across the deployment spectrum.



Event Driven

A vast majority of the architecture is event driven thanks to the use of Solace. We have some synchronous request/reply based messaging to satisfy specific use cases, but we try our best to limit those. Event driven thinking requires a different mindset and it took a few iterations of the architecture to fully embrace it. Once we got there, we went all-in and extended the event-driven architecture all the way to the front-end, eliminating all websocket and AJAX code in favor of Solace's JavaScript API. In our architecture, the UI makes only 1 GET call during the initialization phase. All subsequent server communication is purely event driven through Solace.

> **While the Solace JavaScript NPM package is a bit heavy in terms of size, the benefits of an end-to-end consistent messaging system outweigh the extra bytes that need to be downloaded.**

Since the Solace JavaScript API also supports Request/Response calls ala AJAX it provides a single API for all client-server communications.

Embracing a fully event driven architecture imposes requirements around limiting specific message flows to interested subscribers. For example, while market data is published for all instruments, we don't overwhelm the trader UI with all the market data ticks only to discard those that aren't part of the trader's book. Solace PubSub+ allows fine grained filtering on topics ensuring only relevant data reaches interested subscribers.

A Solace topic is 250 byte / delimited string that is simply metadata on the message that does not need to be provisioned which makes it very lightweight and extremely flexible.

For example, a Solace Topic for FX currency may be modelled as	FX/<source>/<currency_pair>/<type>/<tenor>
Resulting in topics with names such as	FX/FXALL/GBPUSD/FWD/6M and FX/BBG/USDINR/SPOT

Solace provides two wildcards: * which matches a single level and > which matches all levels. Using these two wildcards, we can attract different streams of data to the trader UI and/or pricing engines. Using this mechanism, we can use the following topic subscriptions to get the following streams:

All GBP currency pairs:	**All SPOT currencies:**	**All Bloomberg FX:**	**All FX:**
FX/*/GBP*/>	FX/*/*/SPOT	FX/BBG/>	FX/>

Security

As with any financial services application, security is a top priority. To this end, we use industry standard security mechanisms – use of HTTPS, 2 factor authentication where required, IP whitelisting, multiple stateful firewalls and encryption of sensitive information at rest. In addition, we extensively leverage Solace ACLs to control connections and manage publish and subscribe permissions on the bus. By limiting the fundamental ability to publish or subscribe to topics based on the user of the system upfront we avoid authorization and entitlement checking in our services altogether. To achieve this, we model our topic naming and subscriptions in a hierarchical format, embedding authorized paths within the name and use Solace substitution variables. This hierarchy can be visualized as a tree.

> **By using a combination of wildcards in ACL paths, access to parts of the tree can be controlled, for example limiting the ability of a client to operate on a distinct liquidity pool.**

This mechanism has been proven to work across many different entitlement needs though does require some careful thought upfront.

Portability

Our architecture encourages in-memory processing and state management. All system state is maintained in Solace queues, allowing a new instance of the system to be bootstrapped simply using the messages present on Solace queues. In some cases, we use loaders that populate the Solace queues from persistent storage. In some deployments we archive message flows as well. A new environment can be easily spun up and fed messages to replicate another environment. The use of infrastructure automation tooling helps tremendously in this regard – we've used Ansible, Chef/Puppet and cloud provider tools.

Message model

An important aspect of the implementation is our abstraction over the messaging substrate. The aim of the wrapper API is to decouple the application logic from topics and queues, give it a simple recovery model and a few options on how to do HA. The code path through it is relatively short and lock free with a few atomic references to deal with field visibility between threads and a latch is used during recovery state transitions. In the end, **the same, unaltered application logic occurs during normal message processing and recovery to build an applications state with just some localized pieces of code sensitive to recovery and group membership.**

Simplicity

As evidenced, the architecture leverages integrated, native features provided by the Solace PubSub+ event broker. **The application of our techniques and patterns eliminates the need to integrate additional tools and frameworks as each data flow, access pattern and message distribution mechanism is easily supported on top of Solace.**

5 Conclusion

Building trading platforms can be easy but building a performant, robust, scalable and flexible one is exceeding difficult. We've benefitted tremendously from using Solace – reduced our time to market, enhanced developer productivity, achieved superior system performance, resiliency and global scaling.

Our architecture and patterns have been battle hardened over several live deployments and has been proven to be flexible to adapt to new asset classes and trading modalities.

Do you need an overhaul of your trading system – in whole or parts? Have you been struggling to achieve reference guidelines with your trading system? Call the experts at 28Stone to learn more on how we can help.

About 28Stone

At 28Stone, we leverage our deep capital markets business knowledge, unique technology expertise and proven delivery methodology to help our clients improve their business performance while reducing system risk, cost and time to market. Our consultants partner with their counterparts at the world's leading financial institutions to design and build robust financial applications on time and on budget. At 28Stone, we have a proven track record of quality delivery resulting in the majority of our revenue and growth coming from ongoing, repeat engagements. Learn about us at 28stone.com.

About Solace

Solace helps large enterprises become modern and real-time by giving them everything they need to make their business operations and customer interactions event-driven. With PubSub+, the market's first and only event management platform, the company provides a comprehensive way to create, document, discover and stream events from where they are produced to where they need to be consumed – securely, reliably, quickly, and guaranteed. Behind Solace technology is the world's leading group of data movement experts, with nearly 20 years of experience helping global enterprises solve some of the most demanding challenges in a variety of industries – from capital markets, retail, and gaming to space, aviation, and automotive. Established enterprises such as SAP, Barclays and the Royal Bank of Canada, multinational automobile manufacturers such as Renault and Groupe PSA, and industry disruptors such as Jio use Solace's event broker technologies to modernize legacy applications, deploy modern microservices, and build an event mesh to support their hybrid cloud, multi-cloud and IoT architectures. Learn more at Solace.com.

Have questions or need help?

Contact David Lustig on +1 908 907 1030 to find out how we can help.